NOZOMI NETWORKS

C++ Micro-Problems

Davide Di Gennaro

Software Engineer

C++ Micro-Problems

- Herb Sutter (2013) “My Favorite C++ 10-Liner” https://channel9.msdn.com/Events/GoingNative/2013/My-Favorite-Cpp-10-Liner
- Indeed some (*) problems can be solved by 10 lines of C++
- We will show you three of them

- (*) Not *all* problems unfortunately, but we are working on it

SLIDEWARE

- All the problems described here admit endless variations
- Give a working solution to a problem, but not necessarily complete
- Some important details/optimizations have been intentionally omitted

Global Parameters

Problem:

- There is a global variable that rarely changes
- We need to read it often, from multiple threads

- assume it's not a basic type, otherwise `std::atomic` suffices

Global Parameters

Solution #1

```
mutex m;
string theName;

void SetTheName(const string& s)
{
  lock_guard g(m);
  theName = s;
}

string GetTheName()
{
  lock_guard g(m);
  return theName;
}
```

Global Parameters

Solution #1

- Simple but not super-efficient
  - Locks everytime
  - Copies the string everytime

Ok, Let’s try another approach…

Global Parameters

Solution #2

```
#include <list>
list<string> theHistory{ "" };

void SetTheName(const string& s)
{
  theHistory.push_front(s);
}

const string& GetTheName()
{
  return theHistory.front();
}
```

Global Parameters

Solution #2

- Interesting approach…
  - Not minimally thread safe!
  - We removed the copy but we now have no way of deleting old values

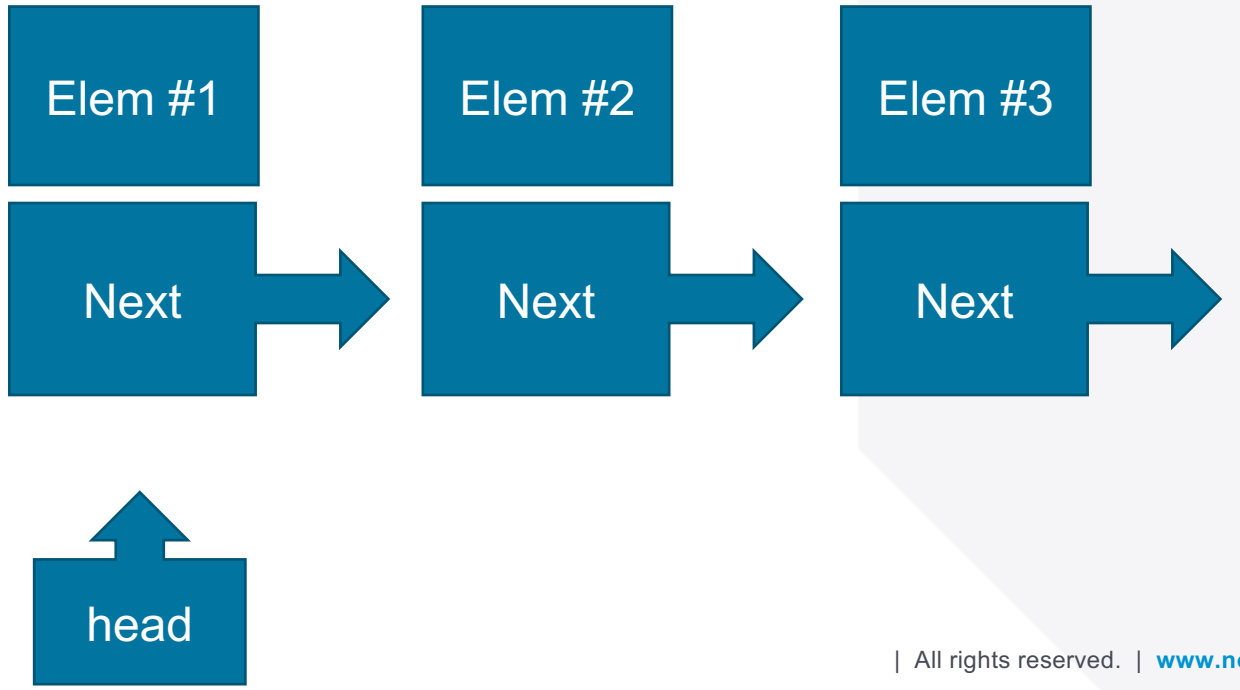
Global Parameters

Solution #2

- Interesting approach… but we can do better
  - Not minimally thread safe!
  - We removed the copy but we now have no way of deleting old values

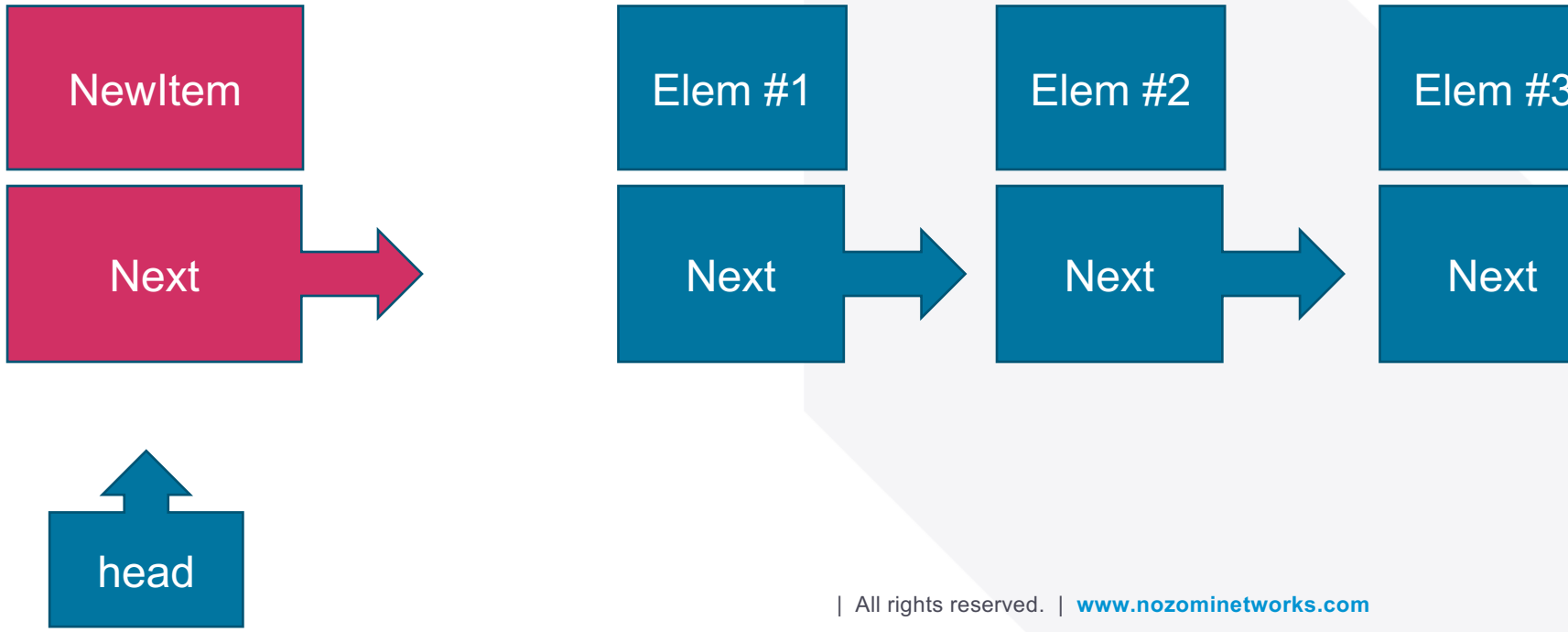
- Under these assumptions we can use an **atomic** linked list

Global Parameters



Global Parameters

- To insert at front, **atomically-swap head and head➔next**



Global Parameters

```
#include "atomic_list.h"
AtomicList<string> theHistory;

void SetTheName(const string& s)
{
  theHistory.PushFront(s);
}

const string& GetTheName()
{
  return theHistory.Front();
}
```

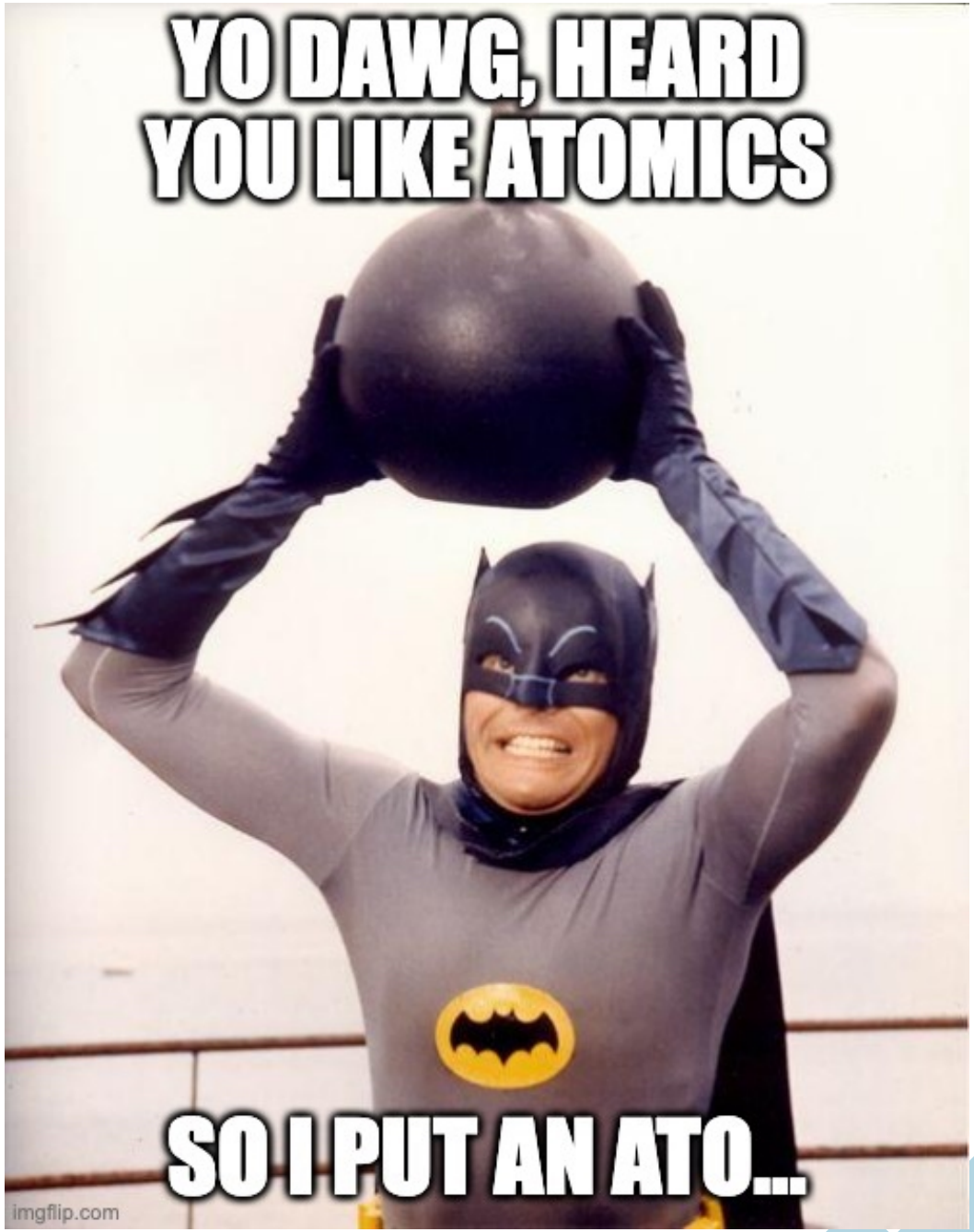
Global Parameters

```
template <typename T>
class AtomicList
{
  struct Node
  {
    T value;
    std::atomic<Node*> next {nullptr};
  };

  std::atomic<Node*> head_ {nullptr};

public:

  template <typename U>
  void PushFront(U&& x);

  const T& Front();
};
```

Global Parameters

atomic_list.h

```
template <typename U>
void PushFront(U&& x)
{
  Node* const n = new Node {std::forward<U>(x), nullptr};
  Node* h = head_.load();
  do
  {
    n->next = h;
  } while (!head_.compare_exchange_weak(h, n));
}

const T& Front()
{
  static const T empty {};
  if (auto s = head_.load())
  {
    return s->value;
  }
  else
  {
    return empty;
  }
}
```


YO DAWG, HEARD
YOU LIKE ATOMICS
SO I PUT AN ATO...
imgflip.com

Global Dictionary

Problem:

- we have a large number of strings but the set of different values is small

Example:

```
class Node
{
    std::string vendor_;
```

- Remember that an empty string uses ~ 24 bytes
- If 95% of nodes have vendor “”, the set of vendors qualifies as small

Global Dictionary

- String Interning
- A technique made popular by Java
- String instances are added to a global dictionary and a pointer to the dictionary entry is used instead
- Bonus: can test equality very fast

Global Dictionary

- String Interning
- A technique made popular by Java
- String instances are added to a global dictionary and a pointer to the dictionary entry is used instead
- Bonus: can test equality very fast

GlobalString.h

```
template <typename T>
struct GlobalDictionary
{
  using dictionary_t = std::unordered_set<std::string>;

  static const std::string* Intern(const std::string& s)
  {
    static dictionary_t words;
    static std::mutex m;

    std::lock_guard g(m);
    return &(*words.insert(s).first);
  }

  static const std::string* Empty()
  {
    static const std::string e;
    return &e;
  }
};
```

Pass a string

exchange it for a pointer into dictionary

Global Dictionary

- String Interning
- A technique made popular by Java
- String instances are added to a global dictionary and a pointer to the dictionary entry is used instead
- Bonus: can test equality very fast
- The pointer is wrapped in a pseudo-string object

GlobalString.h

```
class GlobalString
{
  const std::string* str_;
public:
  GlobalString()
    : str_(GlobalDictionary<void>::Empty())
  {
  }

  GlobalString(const std::string& s)
    : str_(GlobalDictionary<void>::Intern(s))
  {
  }

  // ...
```

Global Dictionary

- String Interning
- A technique made popular by Java
- String instances are global dictionary and a pointer to the dictionary entry is used instead
- Bonus: can test equality very fast
- The pointer is wrapped in a RAII-style object

GlobalString.h

```
class GlobalString
{
  const std::string* str_;
public:
  GlobalString()
    : str_(GlobalDictionary<void>::Empty())
  {
  }

  GlobalString(const std::string& s)
    : str_(GlobalDictionary<void>::Intern(s))
  {
  }

  // ...
```

```
bool GlobalString::operator==(const GlobalString that) const
{
  return ptr_ == that.ptr_ || *ptr_ == *that.ptr_;
}
```


Global Dictionary

- String Interning
- A technique made popular by Java
- String instances are added to a global dictionary and a pointer to the dictionary entry is used instead
- Bonus: can test equality very fast
- The pointer is wrapped in a pseudo-string object

- Bonus: use thread_local storage, but global life

GlobalString.h

```
template <typename T>
struct GlobalDictionary
{
  using dictionary_t = std::unordered_set<std::string>;

  template <typename X>
  static X& CreateAnotherFrom(std::list<X>& c)
  {
    static std::mutex m;
    std::lock_guard g(m);
    return c.emplace_back();
  }

  static const std::string* Intern(const std::string& s)
  {
    static std::list<dictionary_t> global;

    static thread_local dictionary_t& words = CreateAnotherFrom(global);
    return &(*words.insert(s).first);
  }

  static const std::string* Empty()
  {
    static const std::string e;
    return &e;
  }
};
```

borrow local storage from a global container


WHICH THREAD, PRECISELY?
imgflip.com

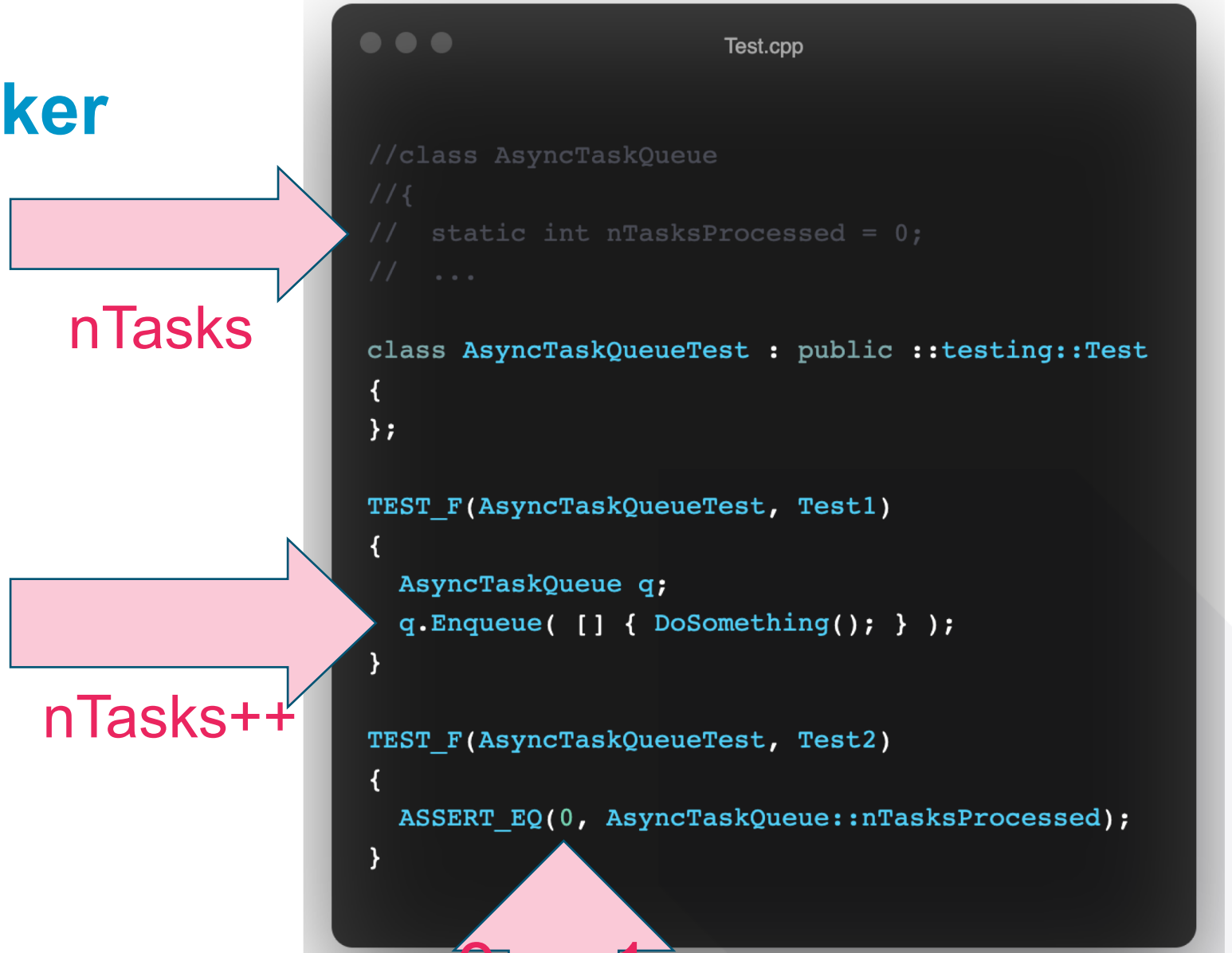
Global Object Tracker

Problem:

- A static variable is initialized when the process starts
- C++ unit tests run in a single process that executes all test functions in some order
- Sometimes a reshuffle makes test fail

Global Object Tracker

Example:



nTasks

nTasks++

0 or 1
depending on the test order

Global Object Tracker

Idea:

- With some metaprogramming tricks:
  - Intercept variable initialization
  - Create a copy of the initializer in a global list
  - When needed, ask the global list to “replay” all the initialitiazions
- Other approaches are possible, but less desirable
  - Balance reasonable coverage/ease of customization
  - Alternative: change the type of the global variable (a wrapper)

G

Ide

- W

- C

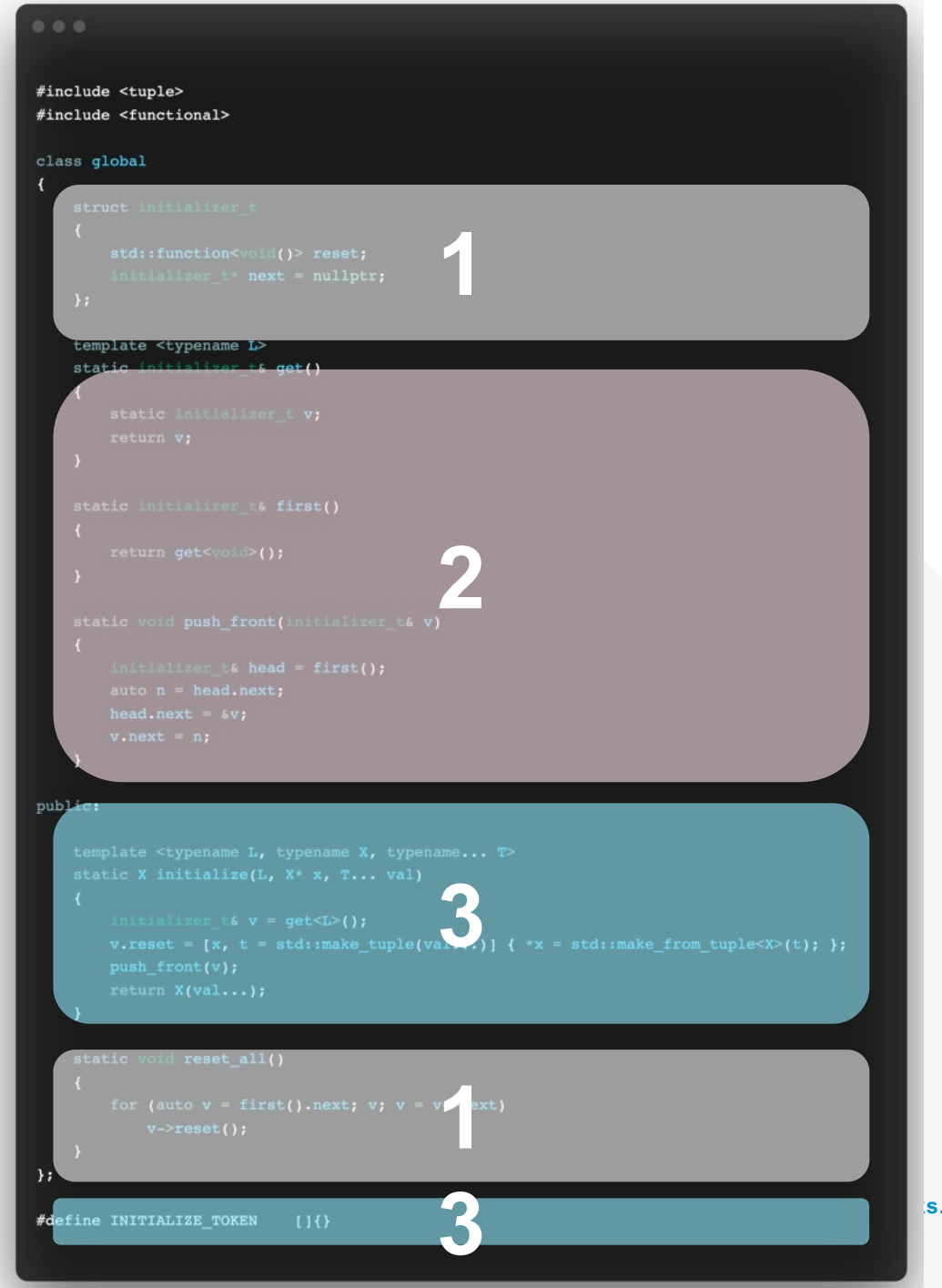
```
#include <string>

std::string s = global::initialize(INITIALIZE_TOKEN, &s, "abc");
std::string t = global::initialize_multi(INITIALIZE_TOKEN, &t, 10, 'x');

int a = global::initialize(INITIALIZE_TOKEN, &a, 7);
int b = global::initialize(INITIALIZE_TOKEN, &b, 8);
int c = global::initialize(INITIALIZE_TOKEN, &c);


int main()
{
    a = 1234;
    b = 789;
    c = -9999;
    s = "blah";
    t.clear();

    global::reset_all();
}
```

Global Object Tracker

We will make use of two unusual language features:

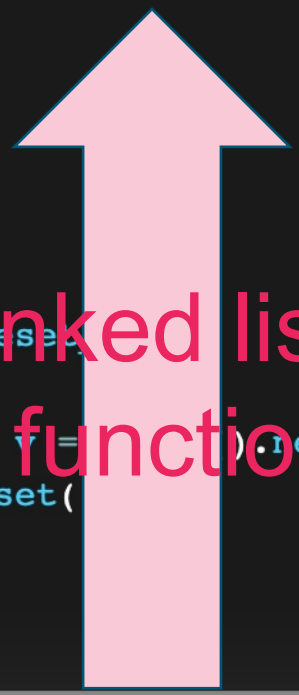
1. The name of a variable is available on the right side of the declaration
   ```
   int n = f(&n);
   ```
2. Multiple instances of the same lambda have different types

Global Object Tracker



Gl

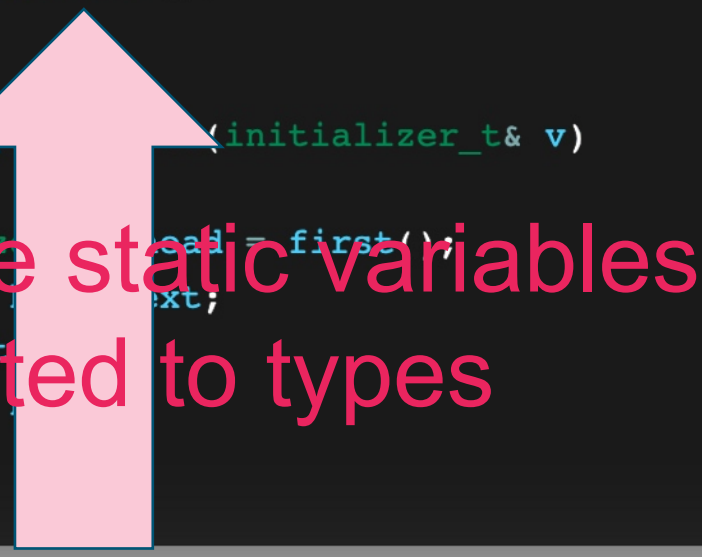
```cpp
#include <tuple>
#include <functional>

class global
{
    struct initializer_t
    {
        std::function<void()> reset;
        initializer_t* next = nullptr;
    };

//...

public:

    static void reset()
    {
        for (auto v = ...next; v; v = v->next)
            v->reset();
    }
};
```



linked list
of functions

Global Object Tracker

```
//...

    template <typename L>
    static initializer_t& get()
    {
        static initializer_t v;
        return v;
    }

    static initializer_t& first()
    {
        return get<void>();
    }

    static void _(initializer_t& v)
    {
        initializ  ead = first();
        auto n = ext;
        head.next
        v.next =
    }
```



list nodes are static variables associated to types

Global Object Tracker

```
public:

    template <typename L, typename X, typename... T>
    static X initialize(L, X* x, T... val)
    {
        initializer_t& v = get<L>();
        v.reset = [x, t = std::make_tuple(val...)] { *x = std::make_from_tuple<X>(t); };
        push_front(v);
        return X(val...);
    }

};

#define INITIALIZE_TOKEN    []{}
```

lambda type L as tag

Global Object Tracker – What I didn't tell you

- This implementation is just the shortest
  - Non-intrusive
  - Good compromise between compactness/specialization/customization

- How to improve:
  - Change `push_front` into `push_back`
  - Write another `initialize` for the default constructor
    - Instead of `*x = X{}`, try calling `x->clear()`
  - Accept a lambda as initializer:
    - `std::string s = global::initialize([] { return "abc"; }, &s);`

Global Object Tracker – What I didn't tell you

- This implementation is just the shortest
  - Non-intrusive
  - Good compromise between compactness/specialization/customization
- How to improve:
  - Change `push_front` into `push_back`
  - Write another `initialize` for the default constructor
    - Instead of `x = X{};`, try calling `x->clear()`
  - Accept a lambda as initializer:
    - `std::string& s = global::initialize([] { return "abc"; }, &s);`

global.h

```
template <typename X>
static decltype(std::declval<X>().clear()) clear(X& x, X*)
{
    return x.clear();
}

template <typename X>
static void clear(X& x, void*)
{
    x = X{};
}

template <typename X>
static void reset(X& x)
{
    clear(x, &x);
}

template <typename X>
static void reset(std::atomic<X>& x)
{
    x = X{};
}
```

Globa

- This im
  - Non
  - Goo
- How to
  - Cha
  - Writ
    - I
  - Acc
    - s

```
template <typename L, typename X, typename... T>
static X initialize(L lambda, X* x, T... val)
{
    initializer_t& v = get<L>();
    push_front(v);

    if constexpr (sizeof...(T) == 0)
    {
        if constexpr (std::is_invocable_r_v<X, L>)
        {
            v.reset = [x, lambda] { *x = lambda(); };
            return lambda();
        }
        else
        {
            v.reset = [x] { reset(*x); };
            return X{};
        }
    }
    else
    {
        v.reset = [x, t = std::make_tuple(val...)] { *x = std::make_from_tuple<X>(t); };
        return X(val...);
    }
}
```

MY BIG RED BUTTON

IS BIGGER THAN YOURS

NOZOMI NETWORKS

Thank You!

Nozomi Networks accelerates digital transformation by protecting the world's critical infrastructure, industrial and government organizations from cyber threats. Our solution delivers exceptional network and asset visibility, threat detection, and insights for OT and IoT environments. Customers rely on us to minimize risk and complexity while maximizing operational resilience.

nozominetworks.com